Section VII -- Programming Techniques

This section discusses programming techniques which will result in substantial performance improvement. It is hardly exhaustive; common sense and elementary matters are not included (e.g.,

A = B * B, not A = B**2.0)

Fortran is used as the demonstration language. Many of these points apply to other languages, though the programmer frequently does not have as much control over them. Fortran is the language of choice for serious development work under VMS when done by professional programmers:

- it tends to be portable
- it has high level constructs, for when their use is tolerable
- it is a low level language, so it can be controlled to operate efficiently
- among "standard" languages, none is more productive for the skilled programmer and most are worse (BASIC and COBOL in particular, including use for DP applications)
- most VMS services can be accessed easily
- it is as complete a language as can be found
- it tends to be more flexible than full high level languages (COBOL, BASIC, PL/I)
- it promotes structured code, but allows deviations from strict structural rules when such are appropriate
- data structures are under the programmer's control, a must for good virtual performance
- the VMS compiler produces relatively good object code.

The new VMS V4 Fortran compiler has an improved optimiser which will produce improved code in many cases (on average, expect 5%). However, there will be cases where it won't, and in some cases the V3 compiler is better (particularly for complex logic flows as opposed to looping and indexing). Bugs have been noted in its compiled output with the only workaround being to compile with the /NOOPTIMIZE qualifier. This results in code which performs terribly.

A major programming concern is to reduce the amount of memory used:

Use smaller structures when possible -- I*2, BYTE -- if values to be stored will fit. For one byte strings, use BYTE, not CHARACTER*1. Exceptions:

- if values will be used as array indexes, use I*4

- if a location is frequently tested as a logical value, use LOGICAL*4, but not if the value is frequently set and reset and only tested in one or two places or infrequently.

Make sure multi-dimensional array filling and referencing is done primarily along the innermost index. Avoid data structures which are filled in a scattered, non-clustered pattern.

Use the new record definition facility to create tight table formats for tables with multiple data types. Elements which are used to make referencing decisions should be separated into parallel tables.

Keep all elements referenced by a particular routine or program phase in a physically compact memory area. Organize common blocks carefully to acheive this. This technique also aids debugging.

Execution speed is increased if code length is shortened:

Use logical tests wherever possible instead of zero/non-zero flags. Bit flag lists are helpful, but don't use the intrinsic functions ISHFT, IBSET, BTEST, IBCLR or ISHFTC as they are implemented by actual subroutine calls, not in-line code. Use:

```
        IF (FLAG/4) THEN
```
instead of
```
        IF (BTEST(FLAG,2)) THEN
```
(but be careful of negative values).

Keep all frequently accessed scalars and small arrays within 127 bytes of the start of a common block or $LOCAL. To control placement place variables in explicitly organized common blocks and use the cross-reference listing to verify offsets. This allows one byte operand offsets instead of 2 or 4.

Minimize argument lists on subroutine calls except where:

- the actual arguments are actually different variables from call to call

- a dimension can be cut out of array referencing (ie., the subroutine confines itself to work on only one vector or plane of an array.) Note that in some cases the compiler may recognize this and optimize for it.

Collect subroutines normally called during the same phase of processing, but not calling each other, into one routine using ENTRY statements. Avoid multi-purpose entry points where a purpose selection variable is passed as an argument. Use separate entry points for each function.

For execution speed in general:

Avoid short subroutines and statement functions that won't be called in line. (See page 1-9, Fortran User's Guide.) A subroutine call and return requires a minimum of 25 microseconds (780). Duplicate code where necessary.

Be careful of multiple element "IF" tests. The onject code produced by Fortran evaluates them from the last element to the first, except where they are different levels of complexity, in which case the simplest are evaluated first. This is contrary to ANSI standards and natural expectations.

Table searches are normally a major user of compute time resources. Adapt techniques which take advantage of natural data or reference ordering. No single "hi tech" search routine is best for all cases. Sort data externally if it reduces searching.

Simple string to numeric conversions (or numeric to string) should be done directly or with the library routines (OTS$CVT_xx_xx). Do not use Fortran internal reads and writes.

Avoid use of LIB$GET_VM, as it is very slow. There is no processing cost if large arrays are declared in the code but not used during execution (assuming the portion that is used is effectively clustered). Calculating addresses at run time can be very time consuming.

Avoid dynamic storage declarations in any form. Turn off bounds checking for production operation.

Avoid tortured code constructions to adhere to the "rules" of structured programming (such as "never" using a GO TO.) The real world has chosen not to conform to the rigid structure acadamicians would like it to have. As a guideline, structured programming is useful; as a religion, it is debilitating.

Avoid designs which call for sub-process creation. (The DCL command "SPAWN" should be outlawed from general usage.) Sub-process creation and many forms of inter-process communication are very expensive and rarely necessary. Under VMS (and easily accessible in Fortran) are any number of features which make parallel processing logic very easy to implement within a single process.

If you must have multiple processes, inter-process control and communication should be handled via

CEF's, AST's and global areas. The lock manager and mailboxes should be avoided.

DCL is inefficient -- rewrite frequently executed routines in Fortran. When DCL is used, eliminate comments, except at the end of the file after a $EXIT line. Minimize @ procedure references -- merge the commands in. Avoid repetitive calls to the same function. Eg.,, don't do

```
        $DELETE A.A;
        $DELETE B.B;
    rather
        $DELETE A.A;,B.B;
```

If you must use Fortran I/O:

Avoid formats (except "(A)" or "(Q,A)") wherever possible. Of course, for report generation this high level facility is invaluable and would only be replaced in intensive, repetitive situations, and then with direct calls to QIO.

Specify RECORDTYPE='FIXED' whenever possible. When doing unformatted I/O, if it can't be fixed, specify 'VARIABLE'. Use the default for unformatted only when doing large "dumps" of data in image form.

Data "lists" in an I/O statement should always consist of exactly one variable, preferably a charcter variable. Implied do lists are as bad as lists of variables.

MACRO

The Fortran compiler is good, but replacing a small compute intensive section of code with well written Macro can often cut execution times by 50%.

Use non-sharable code for maximum speed -- place local data in the same Psect as the code (make it writable) within 32000 bytes, or, preferably, 127 bytes of where it will be used. This allows one or two byte operand offsets without tying up precious registers with base addresses.

When using character instructions, be aware of the values left in registers 0 through 5 after instruction completion. They are very often useful.

Avoid CALLG and CALLS calls to subprogram segments -- use BSBB, BSBW or JSB whenever possible. Avoid POPR and PUSHR if possible. I.e., use registers consistantly.

Frequent calls to system services should not use the macros -- define the argument list explicitly (and locally, if possible) and initialize invariant arguments at compile time.

Explore the VAX instruction set and addressing modes and use them. There is exceptional power there which high level languages just can't take advantage of.

